Provably Efficient Algorithms for Placement of Service Function Chains with Ordering Constraints

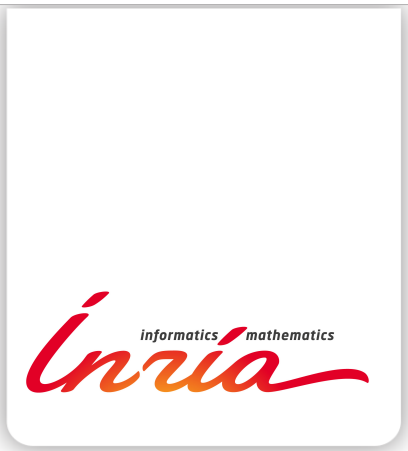
Andrea Tomassilli, Frédéric Giroire, Nicolas Huin, Stéphane Pérennes

Provably Efficient Algorithms for Placement of Service Function Chains with Ordering Constraints

Andrea Tomassilli, Frédéric Giroire, Nicolas Huin, Stéphane Pérennes

RESEARCH REPORT
N° 9141
January 2018
Project-Team Coati

ISSN 0249-6399 ISRN INRIA/RR--9141--FR+ENG



Provably Efficient Algorithms for Placement of Service Function Chains with Ordering Constraints

Andrea Tomassilli, Frédéric Giroire, Nicolas Huin, Stéphane Pérennes

Project-Team Coati

Research Report n° 9141 — January 2018 — 24 pages

Abstract: A Service Function Chain (SFC) is an ordered sequence of network functions, such as load balancing, content filtering, and firewall. With the Network Function Virtualization (NFV) paradigm, network functions can be deployed as pieces of software on generic hardware, leading to a flexibility of network service composition. Along with its benefits, NFV brings several challenges to network operators, such as the placement of virtual network functions. In this paper, we study the problem of how to optimally place the network functions within the network in order to satisfy all the SFC requirements of the flows. Our optimization task is to minimize the total deployment cost.

We show that the problem can be seen as an instance of the Set Cover Problem, even in the case of *ordered sequences* of network functions. It allows us to propose two *logarithmic factor approximation algorithms* which have the best possible asymptotic factor. Further, we devise an optimal algorithm for tree topologies. Finally, we evaluate the performances of our proposed algorithms through extensive simulations. We demonstrate that near-optimal solutions can be found with our approach.

Key-words: network function virtualization, service function chaining, placement, approximation algorithms

This work has been partially supported by ANR program ANR-11-LABX-0031-01, associated Inria Team EfDyNet.

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Algorithmes d'approximation pour le placement de chaînes de fonctions de services avec des contraintes d'ordre

Résumé : Le modèle des réseaux programmables (Software Defined Networks), permet de centraliser la gestion du réseau sur un ou plusieurs contrôleurs et par conséquent de découpler la fonction de contrôle des flux de données. Ce paradigme permet aux opérateurs de réseaux de télécommunications d'offrir des services réseaux complexes et flexibles. Un service se modèlise alors comme une chaîne de fonctions réseaux (firewall, compression, contrôle parental ...) qui doivent être appliquées séquentiellement à un flot de données. Dans cet article, nous étudions le problème du placement de fonctions de services qui consiste à determiner sur quels noeuds localiser les fonctions afin de satisfaire toutes les demandes de service, de façon à minimiser le coût de déploiement.

Nous montrons que le problème peut être ramené à un problème de Set Cover, même dans le cas de séquences ordonnées de fonctions réseau. Cela nous permet de proposer deux algorithmes d'approximation à facteur logarithmique, ce qui est le meilleur facteur possible. De plus, nous proposons un algorithme optimal dans le cas particulier ou la topologie des demandes est un arbre. Finalement, nous évaluons les performances de nos algorithmes par simulations. Nous montrons ainsi qu'en pratique, des solutions presque optimales peuvent être trouvées avec notre approche.

Mots-clés : Virtualisation des fonctions réseaux, Chaînes de fonctions de service, Réseaux logiciels, Optimisation

Contents

1 Introduction **4**

2 Related Work **5**

3 System Model and Problem Formulation **5**

4 Approximation Algorithms for SFC-Placement **6**
4.1 Preliminaries: Single Function. 6
4.2 Equivalence with Hitting Set 7
4.3 Naive and Faster Greedy Algorithms 10
4.3.1 An example 11
4.4 An LP-Rounding Approach. 12

5 An Optimal Algorithm for Tree Topologies **14**
5.1 Special Case: Cost uniform over nodes 17

6 Experimental Study **19**

7 Conclusion and Future Work **22**

1 Introduction

Network Function Virtualization (NFV) is an emerging approach in which network functions are no longer executed by proprietary software appliances but instead, can run on generic-purpose servers located in small cloud nodes [1]. Examples of network functions include firewalls, load balancing, content filtering, and deep packet inspection. This technology aims at dealing with the major problems of today's enterprise middlebox infrastructure, such as cost, capacity rigidity, management complexity, and failures [2]. One of the main advantages of this approach is that Virtual Network Functions (VNFs) can be instantiated and scaled on demand without the need of installing new equipment.
Network flows are often required to be processed by an ordered sequence of network functions. For instance, an Intrusion Detection System may need to inspect the packet before compression or encryption are performed. Moreover, different customers can have different requirements in terms of the sequence of network functions to be performed [3]. This notion is known as Service Function Chaining (SFC) [4].
The same virtual function can be replicated and executed on several servers. It follows that a fundamental problem arising when dealing with chains of network functions is how to map these functions to nodes (servers) in the network while achieving a specific objective. In this paper, we address the problem of how to optimally place virtual functions within the physical network in order to satisfy the SFC requirements of all the network flows. The network is specified by a set of nodes V and links E. The traffic is given as a set of demands $\mathcal{D}$. Each demand is associated with an ordered sequence of network functions that need to be performed to all the packets belonging to the same flow. Our goal is to place network functions reducing the overall deployment or setup cost. The cost aims at reflecting the cost of having a virtual machine that runs a virtual function, such as license fees, network efficiency, or energy consumption [5]. In our framework, we consider a general cost function that depends on both the network node and the network function. We refer to this problem as the *SFC Placement Problem*.
In the case in which all the service chains consist of only one function, the problem is known to be equivalent to the Minimum Set Cover problem, as shown in [6]. This implies that the problem is NP-hard and that an algorithm cannot achieve a better approximation factor than $(1-\varepsilon)\ln|S|$ for any $\varepsilon > 0$, where S is the set of elements to be covered (unless P=NP) [7]. No positive results are known when the lengths of the service function chains are larger than 1.
In this paper, we demonstrate that also the generic case, in which the demands have order constraints on the network functions, also corresponds to a set cover instance. We show that the exponential (in $|V|$) number of sets in the instance can be reduced to a polynomial number (in $|V|$ and $|\mathcal{D}|$) by exploiting the structure of the specific type of set cover instances. It allows us to propose two efficient algorithms for the *SFC Placement Problem*. The first one is based on LP rounding. The second one is a greedy algorithm. For both, we exploit the specific structure of the problem to achieve a short running time, i.e., polynomial also in the length of the largest chain. We show that both the algorithms achieve a solution of cost within a logarithmic factor of the optimal.
We then restrict our attention to tree network topologies. We first show that the problem is NP-hard even in this restricted case. Then, we investigate the scenario in which all the flows are either upstream or downstream flows. We devise an optimal algorithm for this particular case using the dynamic programming technique.
We implement our algorithms and compare their results with the optimal solutions obtained by a linear program. We show that the logarithmic approximation factor is only a worst case upper bound and that we can achieve solutions close to the optimal in most cases.
Although many works on VNF placement have been reported in the literature, no existing work

provides algorithms with proven theoretical results for the placement of chains of VNFs with ordering constraints. Most of the solutions are ILP-based, lacking in scalability, or heuristic-based, with no approximation guarantees. To the best of our knowledge, *we are the first to propose a provably efficient algorithm to place chains of virtualized network functions within the network.*
The rest of this paper is organized as follows. In Section 2, we review related works in more detail. In Section 3, we present the problem formulation. In Section 4, we first show that the *SFC Placement Problem* is equivalent to Set Cover even in the general case. We then present details and analysis of our placement algorithms. In Section 5, we propose our optimal algorithm for tree topologies. In Section 6, we evaluate our proposed algorithms. Conclusions are drawn in Section 7, together with open questions for future work.

2 Related Work

There have been some studies on how to place ordered chains of network functions within the network in the literature. Existing placement algorithms can be roughly classified into two categories: ILP-based and greedy-based. These approaches typically have no provable performance guarantees.
In [8], the authors address the problem of placing and chaining virtual network functions on physical infrastructures minimizing their number. They propose an Integer Linear Programming and a heuristic procedure. The work in [9] studies the joint problem of VNF placement and path selection to better utilize the network. They consider the chaining constraints. Their goal is to maximize the total size of admitted demands. Authors in [10] propose a VNF chaining placement formulated as a Mixed Integer Quadratically Constrained Program. They considered various objectives like minimizing the number of used nodes or the latency of the paths. In [11] and [12], the authors provide both an ILP and a heuristic with resource utilization being their main focus.
The closest works to ours that study the placement of virtual functions as an optimization problem and provide theoretical results for the performance of the proposed algorithms are [13] and [14].
[13] addresses the problem of the placement of virtual functions within the physical network. Each demand has a set of required VNFs that need to be executed. The goal of the authors is to minimize the network cost, given by the setup cost of installing a function on a node and the connection cost that depends on the distance between the clients (i.e., the paths) and the nodes from which they get the service. They provide near-optimal approximation algorithms with theoretically proven performance. However, the execution order of the network functions is not considered in their model.
In [14], the authors focus their attention on the problem of optimal placement and allocation of VNFs to provide a service to all the flows of the network. The goal is to minimize the total number of network functions. In their model, flow routes are fixed, and one flow may be fractionally processed by the same network function at multiple nodes. However, they study the scenario of one single network function and leave the placement of virtual functions with chaining constraint as an open problem for future research.

3 System Model and Problem Formulation

We model the network as a digraph $G = (V, E)$. A demand $d \in \mathcal{D}$ is modeled by a couple composed of a path `path(d)` of length $l(d)$ and a service function chain `sfc(d)` of length $s(d)$. A

$G = (V, E)$	digraph
$\mathcal{D}$	set of demands
$\mathcal{F}$	set of functions
`sfc(d)`	service chain of the demand $d \in \mathcal{D}$
`path(d)`	path associated with the demand $d \in \mathcal{D}$
$l(d)$	length of the path of the demand $d \in \mathcal{D}$
$s(d)$	length of the service chain of the demand $d \in \mathcal{D}$
$c(v, f)$	cost to install the function $f \in \mathcal{F}$ on the node $v \in V$

Table 1: Summary of the notations

path is a sequence of vertices in V. Similarly to [14], we consider the case of an operator which has already routed its demands and which now wants to optimize the placement of network functions. A service function chain is an ordered sequence of functions in $\mathcal{F}$, where $\mathcal{F}$ is the set of network functions. The flow associated with the demand should be processed by the network functions of its chain in the correct order. Each function $f \in \mathcal{F}$ has a setup cost which may depend on the nodes. We note $c(v, f)$ the setup cost of function f in node $v \in V$. In Table 1, we summarize the notations used in this paper.

The problem we consider, referred to as SFC-Placement, is to find a *placement of network functions of minimum setup cost, satisfying the service chain constraints of all demands.* It can be stated as follows.
Input: A digraph $G = (V, E)$, a set of functions $\mathcal{F}$, and a collection $\mathcal{D}$ of demands. Each demand $d \in \mathcal{D}$ is associated with a path `path(d)` $\in V^*$ and to a sequence of functions `sfc(d)` $\in \mathcal{F}^*$. Lastly, a cost $c : V \times F \to c(v, f)$, defining the cost of setting up the function f in node v.
Output: A *function placement* that is a subset $\Pi \subset V \times F$ of function locations, such that, all demands of $\mathcal{D}$ are satisfied. We say that a demand $d \in \mathcal{D}$ associated with a path `path(d)` $= u_1, ..., u_{l(d)}$ and to a chain `sfc(d)` $= r_1, ..., r_{s(d)}$ is *satisfied by* Π, if there exists a sequence of indices $i_1 \leq ... \leq i_{s(d)}$, such that $(v_{i_j}, r_j) \in \Pi$, for $1 \leq j \leq s(d)$.
Objective: minimize $\sum_{(v,f)\in\Pi} c(v, f)$

4 Approximation Algorithms for SFC-Placement

After discussing briefly the trivial subcase in which the service chains have length one, we show that the general problem can be modeled as a Set Cover Problem. The instances have an exponential (in $|V|$) number of sets at first. But, we show that this number can be reduced to a polynomial number (in $|V|$ and $|\mathcal{D}|$) by exploiting the specific structure of the problem. We then propose two algorithms with logarithmic (in $|V|$ and $|D|$) approximation factor. Note that the number of sets is still exponential in the maximum size of a service chain, $s_{\max}$, but this number is small in practice [3] and can thus be considered constant in most scenarios. Finally, we discuss the specific structure of the sets to be covered to improve the efficiency of the algorithms.

4.1 Preliminaries: Single Function.

In this paper, we use the hitting set formulation of the Minimum-Weight Set Cover problem (Min-WSC), which is equivalent [15]. The Minimum-Weight Hitting Set Problem (Min-WHS) can be formally defined as follows:
Input: Collection C of subsets of a finite set S.

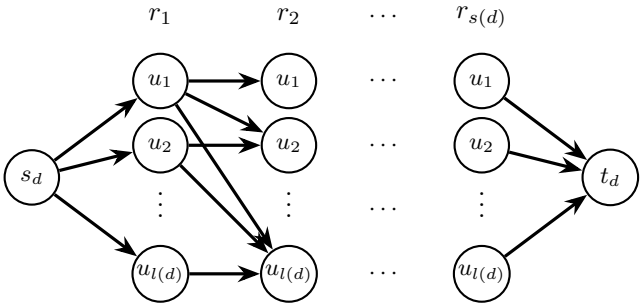


Figure 1: The associated network of a demand $d \in \mathcal{D}$ routed on a path `path(d)` $= u_1, u_2, ..., u_{l(d)}$ that requires a chain `sfc(d)` $= r_1, r_2, ..., r_{s(d)}$

Output: A hitting set for C, i.e., a subset $S' \subseteq S$ such that S' contains at least one element from each subset in C.
Objective: Minimize the cost of the hitting set, i.e., $\sum_{x \in S'} c_x$.

When all the demands have a service function chain which consists of a single function, the problem can be directly mapped to an instance of Min-WHS:
- the elements of S are the possible function locations, i.e., the vertices in V. Each element has cost $c(v)$.
- the sets in C correspond to the paths of the demands in $\mathcal{D}$.
For each path `path(d)`, the corresponding set is the set of all the nodes in the path, i.e., $\{u_1, ..., u_{l(d)}\}$.
The placement of minimum cost covering all demands thus corresponds to a minimum cost hitting set.
In the equivalent Min-WSC formulation, the elements are the paths of the demands and the sets correspond to the function location for node v. The set associated with v has cost $c(v)$ and it is the set of all paths containing v.

The equivalence directly gives us an $H(|\mathcal{D}|)$-approximation using the greedy-algorithm for Set Cover [16] on the positive side. On the negative side, it tells us that the *SFC Placement Problem* is hard to approximate within $\ln |\mathcal{D}|$ [17].

4.2 Equivalence with Hitting Set

We now show that, even in the general case (with order), *SFC Placement Problem* is equivalent to Min-WHS (and so to Min-WSC). For each demand $d \in \mathcal{D}$, we denote with $l(d)$ and $s(d)$ the length of the associated path and chain respectively. Let `path(d)` $= u_1, u_2, ..., u_{l(d)}$ and assume that d requires the sequence of functions $sfc(P) = r_1, r_2, ..., r_{s(d)}$.

Given a demand d, we build an associated network $H(d)$.

Definition 1 (Associated Networks)**.** The network $H(d)$ associated with a demand d is built as follows:

- $H(d)$ has $s(d)$ layers $L_1, L_2, ..., L_{s(d)}$. Each layer contains $l(d)$ nodes corresponding to the nodes of `path(d)`. We note (u_i, j) the i-th node of layer j.
- There is an arc between the node (u, j) and the node $(v, j+1)$ if $u = v$ or if u precedes v in `path(d)`.

- $H(d)$ has two other nodes, s_d and t_d. There is an arc between a node s_d and all the nodes of the first layer and an arc between all the nodes of the last layer and t_d.

See Figure 1 for an example. We then define the capacities to obtain the *capacitated network* $H(d, \Pi)$ associated with a demand d and a function placement Π:

- All arcs have infinite capacity.
- Each node has a capacity, and the capacity of the node u of layer i is 1 if $(u, r_i) \in \Pi$ and 0 otherwise.

Lemma 1. *A demand $d \in \mathcal{D}$ is satisfied by Π if and only if there exists a feasible $st-path$ in the capacitated associated network $H(d, \Pi)$.*

Proof. The intuition of the proof is that an $s_d t_d - path$ (or $st - path$ in short) in the layered graph contains exactly one node from each layer and defines where the flow associated with the demand is going to be processed by the required functions in the specified order. Each layer is associated with a function - the j^{th} layer corresponds to the j^{th} function of the function chain $\texttt{sfc(d)} = r_1, r_2, ..., r_{s(d)}$. Since node (u, j) is connected to $(v, j+1)$ if and only if u precedes v in the path `path(d)`, the sequence of functions is performed in the right order when travelling along the path.

Suppose there exists a feasible $st - path$, p. This means that there exists a set of indices $i_1, ..., i_{s(d)}$ such that $p = \{s, u_{i_1}, ..., u_{i_{s(d)}}, t\}$. This implies that the capacity of u_{i_j} is equal to one, i.e., $(u_{i_j}, r_j) \in \Pi$, for all $1 \leq j \leq s(d)$. Since, in the associated network $H(d, \Pi)$, node (u, j) is connected to $(v, j+1)$ if and only if u precedes v in `path(d)`, we have that $i_1 \leq ... \leq i_{s(d)}$. Therefore all functions of `sfc(d)` are placed in the right order with respect to the nodes of `path(d)`, that is, d is satisfied by Π.

Suppose now that d is satisfied by Π. It means that there exists a set of indices $i_1 \leq ... \leq i_{s(d)}$, such that $(u_{i_j}, r_j) \in \Pi$ for all $1 \leq j \leq s(d)$. Nodes (u_{i_j}, j) of the associated network $H(d, \Pi)$ thus have capacity one. Moreover, there is an arc between (u_{i_j}, j) and $(u_{i_{j+1}}, j+1)$ as u_{i_j} precedes $u_{i_{j+1}}$ in `path(d)`. Hence, $\{s, (u_{i_1}, 1), ..., (u_{i_{s(d)}}, s(d)), t\}$ is a feasible $st - path$ in $H(d, \Pi)$. □

With this notion of associated network, we define the following problem,

Problem 1. Hitting-Cut-Problem $(\mathcal{D}, c)$ *is an instance of the Weighted Hitting Set problem where:*

- *the elements are the function locations (u, f), for all $u \in V$ and $f \in \mathcal{F}$. Its cost is $c(u, f)$.*
- *the subsets of the universe correspond to all the st-vertex-cuts of the associated networks $H(d)$ for all $d \in \mathcal{D}$.*

The problem is thus to find the sub-collection S of elements (functions placement) hitting all the subsets (cuts) of the universe of minimum cost.

Proposition 1. Hitting-Cut-Problem $(\mathcal{D}, c)$ *is equivalent to* SFC-Placement $(\mathcal{D}, c)$.

Proof. By construction, a solution S of Hitting-Cut-Problem corresponds to a solution of SFC-Placement of same cost.

Let us show that S is feasible for Hitting-Cut-Problem if and only if it is a feasible solution of SFC-Placement. The proof is direct using Menger's theorem for digraphs [18]. Consider a digraph and two vertices s and t not connected by an arc. The theorem states that the number of $st - paths$ in a digraph is equal to the minimum st-vertex cut.

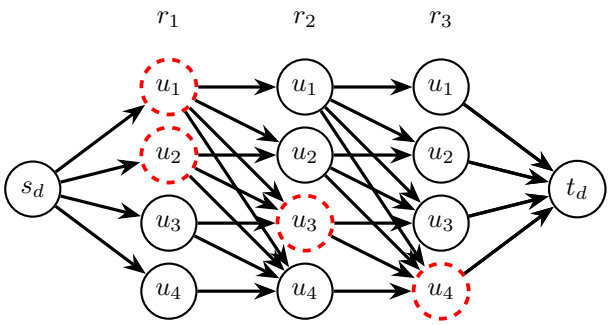


Figure 2: Example of a proper cut (dashed nodes in red) for the layered graph relative to a demand d associated with a path of length 4 and a chain of length 3.

Lemma 1 says that all the demands in $\mathcal{D}$ are satisfied by Π if there exists an $st-path$ in all the associated networks $H(d,\Pi)$ for each $d \in \mathcal{D}$. We thus have that all demands are satisfied if all $st-vertex-cuts$ of $H(P,\Pi)$ have a capacity larger or equal to one. Consider C an st-vertex cut. It is hit by S. This implies that in $H(d,\Pi)$, the capacity of the cut is larger than 1. This yields the proposition. □

Our problem is thus equivalent to a Hitting Set Problem, for which we know approximation algorithms. However, the number of *st*-vertex cuts is exponential in the number of vertices of the digraph. To derive a polynomial algorithm, we need to reduce the size of an instance of Cut-Hitting-Problem. To this end, we use the fact that checking only the *extremal* cuts is enough (An *extremal* cut is a cut that is not strictly included in another cut) and that, in our problem, the extremal cuts of the associated graphs have a specific shape that we call *proper st-cuts*. See Figure 2 for an example.

Definition 2. A proper *st*-cut of the associated graph $H(d)$ is a cut of the following form:

$$\{\underbrace{(u_1,1),...,(u_{j_1},1)}_{\text{layer 1}},\underbrace{(u_{j_1+1},2),...,(u_{j_1+j_2},2)}_{\text{layer 2}},...,$$

$$\underbrace{(u_{j_1+j_2+\cdots+j_{s(d)-1}+1},s(d)),...,(u_{l(d)=j_1+j_2+\cdots+j_{s(d)}},s(d))}_{\text{layer } s(d)}\}$$

for $j_1, j_2, ..., j_{s(d)} \geq 0$, such that $\sum_{i=1}^{s(d)} j_i = l(d)$.

Property 1. *All the extremal cuts of the associated graphs are proper.*

Proof. Let us first remark that, given a cut C in the associated graph, if from the source s it is possible to reach node (u_i,l), then node (u_{i+1},l) can also be reached from the source. Similarly, if the sink t can be reached from node (u_i,l), then the sink can also be reached from node (u_{i-1},l). Suppose that there exists an extremal cut C such that, for a layer l, C contains nodes u_i, u_{i+2} with $u_{i+1} \notin C$. Since by definition C is a cut, we have 2 possibilities:

- u_{i+1} at layer l cannot be reached by the source. Then, all the nodes u_j with $j \leq i+1$ in the layer $l-1$ cannot be reached, and so u_i is not reachable from the source. We can remove it from C and still get a cut. It follows that C is not an extremal cut (contradiction).

- u_{i+1} at layer l cannot reach the sink. In the same way, u_{i+2} cannot reach the sink. We can then remove u_{i+2} from C and still get a cut. C is not an extremal cut (contradiction).

□

Example 1. Consider a demand $D_{a,c}$ that requires the service function chain $\{f_1, f_2\}$. Suppose that the demand is routed on the path $P = \{a, b, c\}$. There are 4 proper cuts: $\{(a, 2), (b, 2), (c, 2)\}$, $\{(a, 1), (b, 2), (c, 2)\}$, $\{(a, 1), (b, 1), (c, 2)\}$, $\{(a, 1), (b, 1), (c, 1)\}$ corresponding respectively to $j_1 = 0, ..., l(d)$.

We can thus define a new problem of smaller size.

Problem 2. *We define the problem* Hitting-Proper-Cut-Problem $(\mathcal{D}, c)$ *as the same problem as* Hitting-Cut-Problem $(\mathcal{D}, c)$*, except that the sets to be hit are only the* proper *st-vertex-cuts of the associated networks $H(d)$ for all $d \in \mathcal{D}$.*

Proposition 2. *The problem* SFC-Placement $(\mathcal{D}, c)$ *is equivalent to a Hitting Set Problem with $\sum_{d\in\mathcal{D}} \binom{l(d)+s(d)-1}{s(d)-1}$ sets as an input. If each demand requires at most $s_{\max}$ network functions and is associated with a path of length smaller than $l_{\max}$, then the size of the instance is at most $O(|\mathcal{D}| \cdot (l_{\max})^{s_{\max}-1})$.*

Proof. The proposition follows from previous results. Hitting-Proper-Cut-Problem $(\mathcal{D}, c)$ is equivalent to Hitting-Cut-Problem $(\mathcal{D}, c)$ as it is enough to consider extremal sets of the collection in a Hitting Set Problem and all extremal cuts are proper cuts. SFC-Placement $(\mathcal{D}, c)$ thus is equivalent to Hitting-Proper-Cut-Problem $(\mathcal{D}, c)$.

The size of the ground set of Hitting-Proper-Cut-Problem $(\mathcal{D}, c)$ is the number of proper cuts of all the associated networks. For each path P, the number of proper cuts of $H(P)$ is simply equal to $\binom{l(d)+s(d)-1}{s(d)-1}$.

Indeed, to obtain the indices $j_1, ..., j_{s(d)}$ defining a proper cut, it is sufficient to select $s(d)-1$ numbers between 0 and $l(d)+s(d)-1$. Without loss of generality, we call them $n_1 \leq ... \leq n_{s(d)-1}$. We then take $j_1 = n_1 - 1$, $j_i = n_i - n_{i-1} - 1$ for $2 \leq i \leq s(d) - 1$, and $j_{s(d)} = (l(d) - s(d) - 1) - n_{s(d)-1}$. We have that $\sum_{i=1}^{l(d)} j_i = l(d)$, so the indices define a proper cut. There are $\binom{l(d)+s(d)-1}{s(d)-1}$ ways of choosing $s(d) - 1$ elements in a set with $l(d) + s(d) - 1$ elements. It yields the number of proper cuts. The size of the ground set is thus $\sum_{d\in\mathcal{D}} l(d)^{s(d)-1}$.

Last, we have $\binom{l(d)+s(d)-1}{s(d)-1} = O(l(d)^{s(d)-1})$. This gives that the number of proper cuts over all paths of the set of demands $\mathcal{D}$ is of the order $O(|\mathcal{D}|(l_{\max})^{s_{\max}-1})$. □

Proposition 2 leads us to two approximation algorithms, a greedy one presented in Section 4.3, and one using LP-rounding presented in Section 4.4.

4.3 Naive and Faster Greedy Algorithms

Naive Greedy Algorithm. The naive greedy algorithm is just the classic greedy algorithm for set cover [16]. It consists of a main loop: while there are proper cuts not hit, it selects the function location with the smallest average cost per newly hit proper cut.

When the demands are routed on paths with length at most $l_{\max}$ and require at most $s_{\max}$ functions, the greedy algorithm achieves an approximation ratio equal to $H(\#\text{Proper Cuts}) = H(|\mathcal{D}|l_{\max}^{s_{\max}-1}) \sim \ln(|\mathcal{D}|) + (s_{\max} - 1)\ln(l_{\max})$ [16], where $H(n)$ is the n-th harmonic number.

Problem for large chains. When the number of functions in the service chains is large, the greedy algorithm could become impractical if it is implemented naively. In fact, the greedy

algorithm selects the function location with the smallest average cost per newly hit proper cut. In a naive implementation, it is necessary to generate explicitly all the proper cuts, and this is not practical since, for a demand d, there may be $O(l_{\max}^{s_{\max}-1})$ of such cuts. Indeed, $l_{\max}$ is in the order of the network diameter. As an example, the network *Cogent* [19] that we consider in the experiments, has a diameter of 28. For a chain of length 10, we would have $\binom{37}{9}$ proper cuts. However, since the structure of the proper cuts is very specific, we can take advantage of it, providing a much faster greedy algorithm.

Faster greedy algorithm, SFCFastGreedy. The main idea of the faster greedy algorithm is to avoid generating all proper cuts by showing it is enough to keep track of the *number of not hit proper cuts*. We show here that, by using dynamic programming, this number can be counted in time $O(|\mathcal{D}|l_{\max}^2 s_{\max})$ (instead of $O(|\mathcal{D}|l_{\max}^{s_{\max}})$).

Let us first introduce some notation. For a demand $d = (\texttt{path(d)}, \texttt{sfc(d)})$, a function placement Π can be seen as a matrix A_d with $l(d)$ rows and $s(d)$ columns and for which $A_d[i,j] = 1$ iff $(u_i, r_j) \in \Pi$. We note $A_d[i:j, k:l]$ the submatrix of A_d considering only the rows from i to j and the columns from k to l.

For a demand $d = (\texttt{path(d)}, \texttt{sfc(d)})$ and a function placement Π (or equivalently A_d), we note $N(d)$ the number of proper cuts not hit by A_d. It can be computed using the recursive function $N(r,c)$ defined below. We have $N(d) = N(l(d), s(d))$ with

$$N(r,c) = \mathbb{1}_{i^*(r,c)=0} + \sum_{j_c=0}^{r-i^*(r,c)} N(n-j_c, c-1), \text{if } c \geq 2$$
$$N(r,1) = \mathbb{1}_{i^*(r,c)=0}$$

where $i^*(r,c)$ is defined as follows. We consider the matrix $A_d[1:r, 1:c]$). We consider the ones placed in the last column of the matrix, column c. If there are none, $i^*(r,c) = 0$. Otherwise, $i^*(r,c)$ is the maximum index of such ones, that is, $i^*(r,c) = \max_{0 \leq i \leq l(d)}\{i, \text{ such that } A_d[i,c] = 1\}$.

The explanation of the formula is the following. We carry out a recursion on the columns of $A_d[1:r, 1:c]$. First, if $i^*(r,c) = 0$, the cut $\{(u_1, f_c), ..., (u_r, c)\}$ is not hit. We thus count $\mathbb{1}_{i^*(r,c)=0}$. We then consider all possible values of j_c for the proper cuts (recall that a proper cut is defined by a set of indices $j_1, ..., j_c$). For a not hit proper cut, $j_c \leq l(d) - i^*$. For a possible value of j_c, the number of corresponding not hit proper cuts is equal to the number of not hit proper cuts in the submatrix $A_d[1 : r - j_c, 1 : c-1]$ for a path of length $r - j_c$ and a chain of size $c-1$, that is, $N(r - j_c, c-1, A_d[1 : r - j_c, 1 : c-1])$.

$N(r,c)$ can be computed using dynamic programming, see the function NC of Algorithm 1. We use a table T with r rows and c columns to keep track of the partial results of the computation. Initially, $T(i,1) = \mathbb{1}_{i^*(r,c)=0}$ for $1 \leq i \leq r$.

4.3.1 An example

Consider a demand d with $\texttt{sfc(d)} = f_1, f_2, f_3$ and $\texttt{path(d)} = u_1, u_2, u_3$. Let Π be a potential function placement. $\Pi = \{(u_1, f_1), (u_3, f_2), (u_2, f3)\}$, that is, f_1 is installed on u_1, f_2 on u_3, and f_3 on u_2. All the required functions are placed, but not in the right order. We show that, in this case, some proper cuts of the associated network $H(d, \Pi)$ are not hit. $H(d, \Pi)$ has $\binom{5}{2} = 10$ proper cuts as shown in Proposition 2. We compute here the number of not hit proper cuts from this set without generating them. The matrix A_d associated with the demand and the starting table T in Algorithm 1 would be the following:

$$A_d = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad T = \begin{bmatrix} 0 & - & - \\ 0 & - & - \\ 0 & - & - \end{bmatrix}$$

As $A_d[1,1] = 1$, we have $i^*(3,1) = 1 \neq 0$ (the cut $\{(u_1,1),(u_2,1),(u_3,1)\}$ is hit). Similarly, $i^*(2,1) = 1 \neq 0$ and $i^*(1,1) = 1 \neq 0$. We thus initialize the first column of T with only zeroes.

In order to compute $T(3,3)$ the following steps are necessary ($i^*(3,3) = 1$):
$T(3,3) = T(1,2) + T(2,2) + T(3,2)$
$T(1,2) = 1 + T(1,1) = 1$
$T(2,2) = T(2,1) + T(1,1) + 1 = 1$
$T(3,2) = T(3,1) = 0$
Since $T(3,3) = 2$, we can derive that 2 proper cuts, out of the overall 10 proper cuts of $H(P,\Pi)$, are not hit. Note that this corresponds to the two proper cuts $\{(v_1,f_2),(v_2,f_2)(v_3,f_3)\}$ and $\{(v_1,f_2)(v_2,f_3)(v_3,f_3)\}$. This shows that the order of the functions is not valid.

From this approach, we can derive a faster algorithm with pseudo-code given in Algorithm 1. At each iteration, the algorithm selects the pair (u,f) of minimum cost, i.e., with the smallest average cost per newly hit proper cut. In order to do this, it makes use of the function NC, calling it for each demand and for each pair $(u,f) \in \mathcal{V} \times \mathcal{F}$. The pair of minimum cost is added to the solution Π. Then, the number of remaining proper cuts to be hit is updated. This process is repeated until all the proper cuts are hit.

Algorithm Complexity. The number of iterations of the main loop of the algorithm is bounded by $|V||\mathcal{F}|$ as we install a function at each iteration. The complexity of the function $\mathrm{NC}(l(d), s(d), \Pi)$ is of the order $O(l(d)^2 s(d))$. It gives us a complexity of $O(l_{\max}^2 s_{\max} |V|^2 |\mathcal{F}|^2 |\mathcal{D}|)$, when a naive algorithm would be of order $O(l_{\max}^{s_{\max}} |V|^2 |\mathcal{F}|^2 |\mathcal{D}|)$, as it would generate all proper cuts.

4.4 An LP-Rounding Approach.

First formulation. The Hitting-Proper-Cut-Problem can be formulated as an ILP. For each node $u \in V$ and for each function $f \in \mathcal{F}$, we define the decision binary variable $x(u,f)$ that indicates whether the function f is installed on node u ($x(u,f) = 1$ in this case). We get as global ILP:

Objective

$$\min \sum_{u \in V} \sum_{f \in \mathcal{F}} c_{u,f} \cdot x_{u,f}$$

Cover conditions

$$\forall d \in \mathcal{D}, \sum_{(u,f) \in C} x_{u,f} \geq 1, \forall\ C \text{ proper cut of } A(d)$$

We consider here the Set-Cover approximation through LP-Rounding. For each $u \in V$ and $f \in F$, we relax the ILP by replacing the constraints $x(u,f) \in \{0,1\}$ by $0 \leq x(u,f) \leq 1$. The relaxed ILP can be solved in time polynomial in the number of constraints. Let x^* be an optimal solution to the LP relaxation. Each fractional variable $x^*(u,f)$ is rounded to 1 with probability $x^*(u,f)$. The problem is then solved again with the additional constraints given by the rounded variables.

The process is repeated iteratively until all the variables have values in $\{0,1\}$. With this approach, we find a feasible solution with logarithmic approximation ratio in *expected* polynomial time (in the number of constraints) [20]. The number of constraints is the number of proper cuts,

Algorithm 1 SFCFastGreedy

Input: set of demands $\mathcal{D}$
for each $d \in \mathcal{D}$ **do**
    $not\text{-}hit[d] \leftarrow \binom{l(p)+s(P)-1}{s(P)-1}$
$\Pi = \emptyset$
repeat
    $min\_cost \leftarrow +\infty$
    $best\_sol \leftarrow null$
    $best\_not\text{-}hit \leftarrow null$
    for each $(u, f) \in \mathcal{V} \times \mathcal{F}$ **do**
        $newly\_hit \leftarrow 0$
        $\Pi' \leftarrow \Pi \cup \{(u, f)\}$
        for each $d \in \mathcal{D}$ **do**
            $T = l(d) \times s(d)$ matrix of null
            for $1 \leq i \leq l(d)$ **do** ▷ initialization of T
                $T[i, 1] \leftarrow \mathbb{1}_{i^*(i,1)}$
            $new\text{-}not\text{-}hit[d] \leftarrow \mathrm{NC}(l(d), s(d), \Pi')$
            $newly\_hit$ += $not\text{-}hit[d] - new\text{-}not\text{-}hit[d]$
        $cost \leftarrow \frac{\mathrm{cost}(u,f)}{\mathrm{newly\_hit}}$
        if $cost < min\_cost$ **then**
            $min\_cost \leftarrow cost$
            $best\_sol \leftarrow (u, f)$
            $best\_not\text{-}hit \leftarrow new\text{-}not\text{-}hit$
    $\Pi = \Pi \cup \{best\_sol\}$
    $not\text{-}hit \leftarrow best\_not\text{-}hit$
until $not\text{-}hit[d] = 0$ for each $d \in \mathcal{D}$
Output: placement Π

function NC (row r, column c, Π)
    ▷ Recursive function used to count the number of proper cuts not hit given a demand d and a function placement Π
    if $T[r, c] \neq$ null **then return** $T[r, c]$
    $result \leftarrow 0$
    if $i^*(r, c) = 0$ **then** $result \leftarrow result + 1$
    for $0 \leq j \leq n - i^*(r, c)$ **do**
        $result$ += $\mathrm{NC}(n - j, c - 1)$
    $T[r, c] \leftarrow result$
    return $result$

which is of the order $O(|\mathcal{D}| l_{\max}^{s_{\max}-1})$. It is thus polynomial in $|\mathcal{D}|$, the number of demands, but exponential in $s_{\max}$, the maximum size of a service chain. As discussed, this number is small in practice, but it may still have a strong impact on the algorithm execution time. We propose a faster algorithm below.

Faster rounding algorithm, SFCFastRounding. In fact, similarly as for the greedy algorithm, we can avoid generating explicitly all proper cuts. The idea is to use the formulation of the problem looking for a path in the associated networks $H(d, \Pi)$, as it is equivalent. We derive another ILP formulation. The binary decision variables are now of two kinds:
(i) Location or capacity variables. These variables are the same as in the first formulation: $x(u, f)$ indicates in the first formulation whether the function f is installed on node u. In the second formulation, it corresponds to the shared capacity of the node (u, f) of the associated networks.
(ii) Flow variables. For each demand $d \in \mathcal{D}$, we have a flow variable f_{uv}^d for each edge of the associated network $H(d)$.

The constraints are (i) node capacity constraints and (ii) flow conservation constraints. There are $O(|V| + s_{\max} l_{\max} |D|)$ constraints, a number which is now polynomial in $s_{\max}$.

Objective

$$\min \sum_{u \in V} \sum_{f \in \mathcal{F}} c_{u,f} \cdot x_{u,f}$$

Capacity constraints. $\forall u \in V, \forall f \in \mathcal{F}$,

$$\sum_{d \in \mathcal{D}} \sum_{vu \in E(H(d))} f_{vu} \leq x_{u,f},$$

Flow conservation constraints. $\forall d \in \mathcal{D}$,

$$\sum_{uv \in E(H(d))} f_{uv}^d = \sum_{vu \in E(H(d))} f_{vu}^d, \ \forall \ u \in V(H(d)) \setminus \{s_d, t_d\},$$

$$\sum_{s_d v \in E(H(d))} f_{s_d v}^d = 1$$

A solution of the second formulation corresponds to a solution of the first formulation of same cost (as finding paths in the associated networks is equivalent to covering the cuts, see Lemma 1). Therefore, the rounding can be carried out in the same way and leads to the same approximation factor.

To summarize, along with the fast greedy algorithm, SFCFastGreedy, we obtain a second approximation algorithm for SFC-Placement, called SFCFastRounding, with the same approximation factor $O(\ln(|\mathcal{D}|) + (s_{\max} - 1)\ln(l_{\max}))$. Its expected execution time is $O(M \ln M)$ with $M = |V| + s_{\max} l_{\max} |D|$.

5 An Optimal Algorithm for Tree Topologies

In this section, we restrict our attention to tree logical network topologies. Note that the physical network itself can be of any shape, but the clients are communicating through a tree. The network architecture of today's data centers typically consists of a tree of routing and switching elements [21]. Moreover, tree topologies are widely used, e.g., for Wireless Sensor Networks [22], and Content Delivery Networks [23]. We first prove that the *SFC Placement Problem* is NP-hard even on trees through a reduction from the *Vertex Cover Problem*. Then, for the special case in which all the flows are either upstream or downstream flows (i.e., flows are either going towards the tree root or towards the leaves), we devise an optimal algorithm, TreeSFCAlgo.

Theorem 1. *The SFC Placement Problem is NP-hard even on a tree and in the case of a single network function.*

Proof. Given a graph $G = (V, E)$ and a positive weight function $w : V \to \mathbb{R}^+$, a vertex cover of minimum weight is a subset $C \subseteq V$ such that $\forall\ (u, v) \in E, u \in V$ or $v \in V$ (or both) and $\sum_{u \in C} w(u)$ is minimized.
Let $\mathcal{I} = (G = (V, E), w)$ be an instance of Vertex Cover. We can create an instance $\mathcal{I}'$ of *SFC-tree Placement* by taking the digraph $T = (V \cup \{r\}, \{(u, r), \forall u \in V\} \cup \{(r, u), \forall u \in V\})$. For each $uv \in E$, we create a demand d with $\texttt{path(d)} = u, r, v$ and $\texttt{sfc(d)} = \{f\}$. The setup cost is $c(u, f) = w(u)$ for all $u \in V$, and $c(r) = \sum_{(u) \in V} w(u) + 1$ for the root of the tree. Note that with this choice of costs, the function f is never placed in the root in an optimal placement, as it is cheaper to place the function in all the other vertices of the tree. We thus have the following equivalence: There is a function placement that satisfies all the paths' requirements in the tree with cost at most $\leq c \iff$ G has a vertex cover of cost $\leq c$. The reduction can be done in polynomial time. It only requires scanning all the edges and creating the set of demands $\mathcal{D}$. Since Vertex Cover is NP-hard to approximate within a factor of 1.36 [24], then the Placement Problem cannot be solved in polynomial time even on trees. □

We now provide a polynomial algorithm that computes the optimal solution in the upstream/downstream case. We present the algorithm in the upstream case, since downstream flows can be replaced by upstream flows, by reversing both the paths and the required function chains.
Main idea. We use dynamic programming in a bottom-up fashion. Given a sub-tree T_v rooted at v, we call a *partial solution*, a feasible function placement restricted to T_v. We also distinguish 3 kinds of paths: *internal-paths,* all vertices of the paths are inside T_v; *external-paths,* no vertex is in T_v; and *crossing-paths,* some but not all vertices are in T_v.

In fact, partial solutions can be encoded in a compact way. To see that, we look at how a partial solution s interacts with a global solution and we claim that:

a) s has to cover all the internal paths.

b) s has no impact on the external paths.

c) On each crossing-path, s provides some (potentially empty) prefix of the required function chain.

d) s induces some cost, namely the cost of the functions located inside T_v.

Since a) and b) are common to all partial solutions, a partial solution is fully characterized by (c) and its cost (d). Now to code c), remark that, instead of remembering for each external path what prefix is provided inside T_v, one may keep track of what suffix must be provided outside T_v. Now, since all paths are upstream, we may simply remember that some suffix s must be provided outside T_s at depth $\geq x$. We call this a *constraint*. The key element here is that, if two paths share the same suffix, one only needs to keep the one that stops at the largest depth.

Overall, this means that a partial solution can be encoded with a set of constraints, and its internal cost. So, our algorithm computes inductively for each subtree, the table containing, for each possible list of constraints, the minimum cost of a partial solution matching these constraints.
TreeSFCAlgo. Let us first introduce some notations and definitions, summarized in Table 2. We note $\texttt{depth}(u)$, the depth of a node u in the tree T (the tree root is at depth 1). Let $\mathcal{C}$ be the set of service chains (a chain per demand). We call $\texttt{suff}(\mathcal{C})$ the set of suffixes of elements of $\mathcal{C}$.

Table 2: New definitions and notations.

$\mathcal{D}_u$	the set of demands s.t. `path(d)` starts at Node u
`src`(d)	source of the path `path(d)`
`dest`(d)	destination of the path `path(d)`
$\mathcal{C}$	set of distinct service chains
`suff`$(\mathcal{C})$	set of suffixes of service chains
`depth`(u)	depth of node $u \in V$ in the tree (source is at depth 1)
`deg`(u)	degree of node $u \in V$ in the tree (# children = `deg`-1)
constraint c	couple (chain suffix,destination d_s)
partial solution s	couple (set of constraints C_s,`cost`(s))
table S_u	set of partial solutions of node u

A *constraint* is a couple ($s \in$ `suff`$(\mathcal{C})$, $h \in \mathbb{N}$). A constraint positioned at node u means that the subchain s must be placed in parents of u with depth larger of equal to h. To each demand $d \in \mathcal{D}$ is associated the constraint (`sfc(d)`, `depth`(`dest`(d))), positioned at the node $src(d)$. This means that the chain `sfc(d)` has to be placed below node `dest`(p). Let C_1 and C_2 be two sets of constraints. Two operations may be done to a set of constraints, POP and MERGE.

- MERGE(C_1, C_2). The MERGE operation is a union with "suffixe uniqueness": if $(s, h_1) \in C_1$ and $(s, h_2) \in C_2$, then only $(s, \max(h_1, h_2))$ is present in MERGE(C_1, C_2), as this is the most stringent constraint.

- POP$(F \subseteq \mathcal{F}, C_1)$. We update every suffix σ of C_1 by removing from it the longest prefix made of functions present in F.

A *partial solution* at a node of the tree is encoded by a set of constraints and a cost. A *table* is a set of partial solutions. We note S_u, the table of node u.

- MERGE(S_1, S_2). Two tables S_1 and S_2 may be merged by building a partial solution z for each pair of partial solutions $x \in S_1$ and $y \in S_2$. The constraints of z are the MERGE of the constraints of x and y. The cost of z is just the sum of the costs of x and y. The pseudo-code of all functions and of the algorithm is given in Algorithm 2.

- MERGE$(S_1, ...S_n)$. n tables $S_1, ...S_n$, with $n > 2$, may be merged by doing a two-by-two merge in any order (by associativity of the MERGE function).

We now present our solution TREESFCALGO (pseudo-code in Algorithm 2). It considers the nodes one by one starting from the leaves and builds the tables of each node. S_u, the table of node u is created from intermediate tables $S_{\mathcal{D}_u}$, $S_{\texttt{children}(u)}$, and the tables of its children in the following way. For a node u, it first builds the table $S_{\mathcal{D}_u}$, corresponding to the demands whose paths start in u, using function BUILD_CONSTRAINTS$(\mathcal{D}_u)$. $S_{\mathcal{D}_u}$ contains a single solution of cost 0. The constraints of this solution are built in the following way. For each demand $d \in \mathcal{D}_u$, create the constraint (`sfc(d)`, `depth`(`dest`(d))). Then, it does the MERGE of all the generated constraints. TREESFCALGO then builds $S_{\texttt{children}(u)}$ by merging $S_{\mathcal{D}_u}$ with the tables of its children. Lastly, using function ADD_NODE$(u, \mathcal{D}_u)$, it considers all possible function placements in u and, for each one of them, it considers all solutions in $S_{\texttt{children}(u)}$ and updates the constraints and

cost if the placement is compatible with them, using the POP operation. Updating a constraint means removing the functions placed at node u from the suffix representing the chain functions which remain to be placed.

When the table of the root of T is computed, we can select the best solution. The last step of the algorithm is to reconstruct the solution by doing a second pass on the tree, starting from the root.

Time complexity. TREESFCALGO is doing a loop over the vertices of T:

- Complexity of BUILD_CONSTRAINTS. During the whole algorithm, we consider all demands of $\mathcal{D}$. For each demand d, we build the constraint $(\texttt{sfc(d)}, \texttt{depth}(\texttt{dest}(d)))$. Computing the depth of all nodes can be done beforehand with a single pass on the tree of cost $O(|V|)$. We then check the uniqueness of the constraint in $S_{\mathcal{D}_u}$. The test takes constant time using a hash table. We thus obtain an amortized complexity: $O(|V(T)| + |\mathcal{D}|)$.

- Complexity of MERGE: A table is a set of solutions. The size of a solution $x = (C_s, c)$ is given by its set of constraints. The number of constraints is limited by the number of possible suffixes of chains, $s_{max}|\mathcal{C}|$, where s_{max} is the size of the longest chain. Thus, the memory to store a solution is of order $O(s_{max}|\mathcal{C}|)$. The size of a table is then limited by the number of possible sets of constraints and is thus of order $O(2^{s_{max}|\mathcal{C}|})$.

  Merging two tables of size $O(2^{s_{max}|\mathcal{C}|})$, has a complexity of $O(2^{2s_{max}|\mathcal{C}|})$, as we consider each pair of elements. To merge the tables of u's children, as discussed and due to the associativity of the MERGE function, we can do it iteratively, leading to a complexity of $O(n2^{2s_{max}|\mathcal{C}|})$.

- Complexity of ADD_NODE: For each possible placement of functions of u ($2^{|\mathcal{F}|}$ potential placements), we consider each solution in $S(u)$ ($2^{s_{max}|\mathcal{C}|}$ potential solutions). For each solution, we update its set of constraints ($s_{max}|\mathcal{C}|$ potential constraints). The time to update a constraint: $O(s_{max})$, with s_{max} maxsize of a suffix. This leads to a complexity of $O(s_{max}^2|\mathcal{C}|2^{|\mathcal{F}|+s_{max}|\mathcal{C}|})$.

In summary, we get a complexity of $O(|\mathcal{D}| + |V| + |V|^2 2^{2s_{max}|\mathcal{C}|} + |V|s_{max}^2|\mathcal{C}|2^{|\mathcal{F}|+s_{max}|\mathcal{C}|})$. The number of functions $|\mathcal{F}|$ and the number of chains $|\mathcal{C}|$ are usually small in practice. They can thus be considered constant most of the time. The algorithm is thus *quadratic in the number of nodes of the tree and linear in the number of demands.*

Memory usage. The memory used during the algorithm is to keep the tables for all vertices, that is $O(|V|2^{s_{max}|\mathcal{C}|})$. The memory is thus linear in the number of vertices.

5.1 Special Case: Cost uniform over nodes

When the cost of setting up a function f is the same for each node of the graph ($\forall v, v' \in V, c(v, f) = c(v', f)$), the algorithm can be improved using the following lemma.

Lemma 2. *There exists an optimal solution placing only functions on nodes which are destinations of a path.*

Proof. Consider an optimal solution. We create a new solution in the following way. For each function f placed in a non-destination node u, we move it up in the tree towards the root to the first destination node v encountered. The set of demands satisfied by u is a subset of the set of demands satisfied by v. We thus built a feasible solution. The new solution has the same cost as the first one, as the number of placed functions is the same. □

Algorithm 2 TreeSFCAlgo

Input: T with root r
T'=T
while True **do**
  Consider a leaf u of T'
  $S_{\mathcal{D}_u} \leftarrow$ BUILD_CONSTRAINTS($\mathcal{D}_u$)
  $S_{\texttt{children}(u)} \leftarrow$ MERGE($S_{\mathcal{D}_u}, S_{v_1}, ..., S_{v_n}$), with $v_1, ..., v_n$ the children of u in T
  $S_u \leftarrow$ADD_NODE($u, S_{\texttt{children}(u)}$)
  $T' \leftarrow T' \setminus \{u\}$
Output: return solution of S_r with minimum value.

---

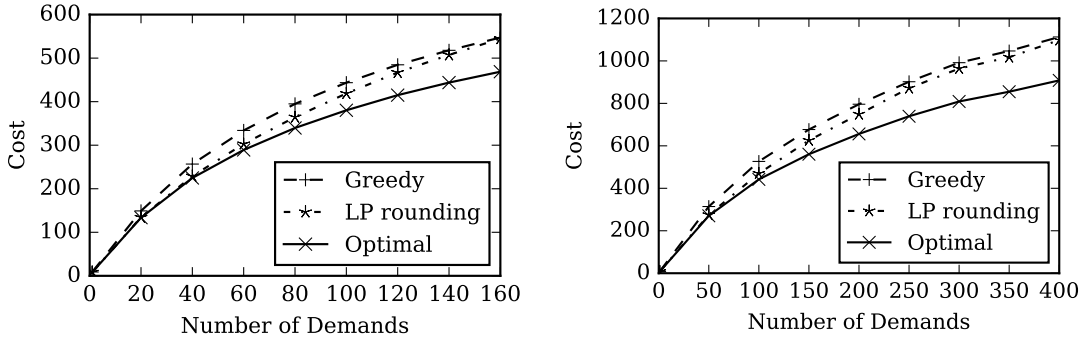
function MERGE(S_1,S_2)
  ▷ Merging two tables S_1 and S_2
  S $\leftarrow$ {}
  for each $x \leftarrow (C_x, cost_x) \in S_1$: **do**
    for each $y \leftarrow (C_y, cost_y) \in S_2$: **do**
      $C_z \leftarrow$MERGE(C_x, C_y)
      $cost_z \leftarrow cost_x + cost_y$
      if $(C_z, c) \notin S$ **then**
        S.append($C_z, cost_z$)
      else
        S.append(C_z, min($cost_z, c$))
  return S

---

function BUILD_CONSTRAINTS($\mathcal{D}_u \subseteq \mathcal{D}$)
  ▷ Building $S_{\mathcal{D}_u}$ from $\mathcal{D}_u$, the set of demands with a path starting in u. For each chain s of a demand in $\mathcal{D}_u$, we keep a constraint with s and the deepest destination of a path with the chain.
  C $\leftarrow$ {}
  for each $d \in \mathcal{D}_u$ **do**
    C$\leftarrow$MERGE(C,{($\texttt{sfc(d)}, \texttt{depth}(\texttt{dest}(d))$)})
  return S$\leftarrow${(C,0)}

---

function ADD_NODE($u, S_{\texttt{children}(u)}$)
  ▷ Build S_u, the table of solutions of node u
  $S_u \leftarrow$ {}
  for each $r \subseteq \mathcal{F}$ **do** ▷ functions installed on node u
    for each $s \leftarrow (C_s, cost(s)) \in S_{\texttt{children}(u)}$ **do**
      if s is compatible with r (meaning if all constraints with level d are satisfied by r) **then**
        $C_s \leftarrow$POP(r, C_s) ▷ update constraints of s
        $cost(s) \leftarrow cost(s) + \sum_{f \in r} c(u, f)$ ▷ update cost
        if $(C_s, c) \notin S$ **then** ▷ ensure uniqueness
          S.append($C_s, cost(s)$)
        else
          S.append(C_s, min($cost(s), c$))
  return S



(a) InternetMCI

(b) germany50

Figure 3: Average setup cost as a function of the number of demands

Tree contraction. Following Lemma 2, the first step of the algorithm is to contract the paths and the tree T by removing the non-destination nodes. We obtain a contracted tree, T^*, and a set of contracted paths, $\mathcal{P}^*$. Note now that all paths of $\mathcal{P}^*$ start at a destination node (either its own destination node or the destination of another path). To each destination node u, we associate the set of contracted paths starting in u, $\mathcal{P}_u$.

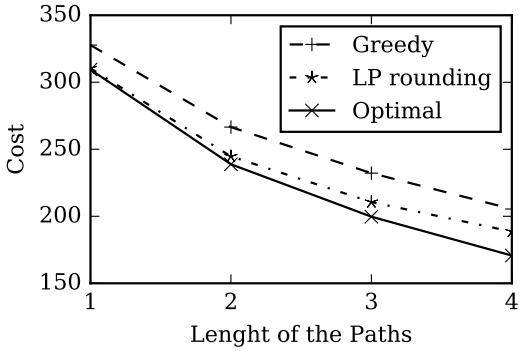
6 Experimental Study

In this section, we evaluate the performances of our proposed algorithms: SFCFastRounding and SFCFastGreedy, referred to as LP rounding and Greedy in the plots, respectively. We study how the total setup cost and the accuracy of our algorithms vary according to four different settings: (i) different path lengths, (ii) increasing number of demands, (iii) increasing length of the service function chains, and (iv) different network topologies. We compare the solutions computed by our algorithms with the optimal ones computed by solving an ILP using IBM ILOG CPLEX.

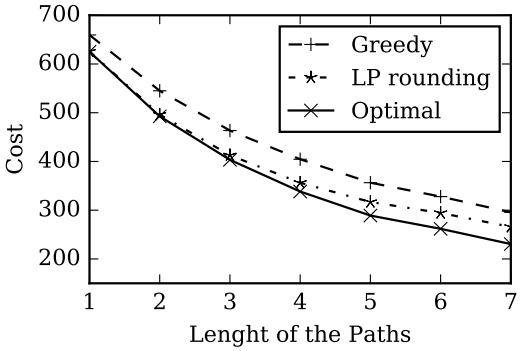
We show that the logarithmic approximation ratio is just a worst case upper bound and that our algorithms perform well in all the considered scenarios. In fact, the additional cost of the solutions computed by the two algorithms never exceeds 25% of the optimal one. Moreover, the LP rounding algorithm usually obtains a better ratio than the greedy one, but at a cost of a much higher processing time.

Data sets. We conduct experiments on two real-world topologies of different sizes: `InternetMCI` [19], (19 nodes and 33 links) and `germany50` [25], (50 nodes and 88 links), and on random Erdős-Rényi graphs [26]. We build our instances in the following way. The source and destination nodes of a demand are uniformly chosen at random from the set of vertices. The path of the demand is given by a shortest path between these two nodes and its chain is composed of 2 to 6 functions uniformly chosen at random from a set of 30 functions. Finally, the setup cost of a function on a node is uniformly chosen at random between 1 and 5.

Number of demands. We first compare the performances of the algorithms in the case of an increasing number of demands. Results are given in Figure 3. In this scenario, we consider up to 160 demands for `InternetMCI` and up to 400 for `germany50`. As expected, we see that the setup cost increases with the number of demands, as the number of functions to be placed increases. However, the increase is sublinear. The reason is that, the more demands in a network, the higher



(a) InternetMCI



(b) germany50

Figure 4: Average setup cost as a function of the length of the paths

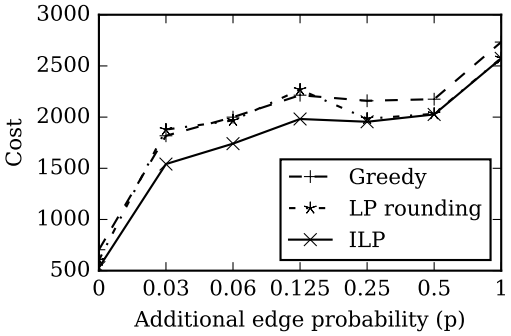


Figure 5: Average setup cost in random graphs as a function of the additional edge probability

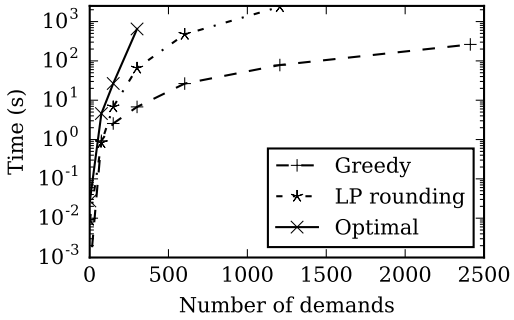
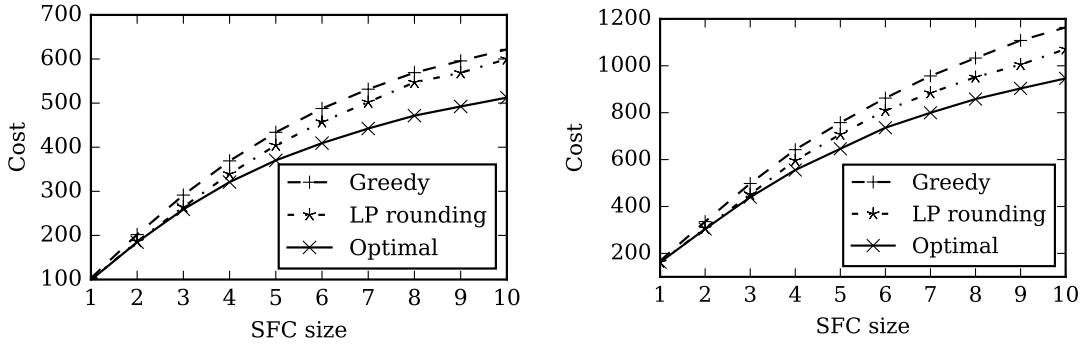


Figure 6: Average completion time as a function of the number of demands on Cogent

the opportunity of sharing functions. The optimality ratio is at most 21% for both algorithms. The solution provided by the greedy algorithm differs from 7 to 15% from the optimal one for `InternetMCI` and from 10 to 21% for `germany50`. However, the LP rounding algorithm shows an interesting behavior. When the number of demands is small, it finds optimal solutions. As the number of demands increases, its accuracy deteriorates faster than the one of the greedy algorithm. For the highest number of demands, both algorithms exhibit similar performance.

Length of the paths. We now study the impact of the length of the paths. We only consider demands with pairs of nodes at equal distances, from 1 to 4 for `InternetMCI`, and from 1 to 7 for `germany50`. For each length, we consider 40 demands for `InternetMCI` and 75 demands for `germany50`. As we can observe in Figure 4, in both networks, the total setup cost strictly decreases when the length of the path increases. In fact, when paths are longer, the demands tend (in average) to share more nodes, reducing the number of required functions to satisfy all the demands and so the cost. For both topologies, the LP rounding algorithm performs better than the greedy one. For the rounding algorithm, the ratio to the optimal solution is smaller than 10% for `InternetMCI` and 15% for german50. The greedy algorithm presents a gap from the optimal solution between 6 ($l(d) = 1$) and 20% ($l(d) = 4$) for `InternetMCI` and between 5



(a) InternetMCI

(b) germany50

Figure 7: Average setup cost with respect to the length of the service function chains

($l(d) = 1$) and 25% ($l(d) = 7$) for `germany50`.

Length of the chain. We now look at the impact of the service function chains' length on the algorithms' accuracies. In this experiment, we consider service function chains, composed of 1 to 10 functions. In total, we route 75 demands for `InternetMCI` and 150 for `germany50`. As shown in Figure 7, an increasing length of the chains impacts the performance of the algorithms negatively. In fact, for `InternetMCI`, the ratio between the solution computed by the LP Rounding algorithm and the optimal solution varies from 0.1% with a single function chain to 17% with 10 functions in the chain. For the greedy algorithm, it ranges from 4 to 21% for chains of length 1 and 10, respectively. We observe the same results for the `germany50` topology. The solution of the LP rounding algorithm varies from 0.1 to 13%, while the solution of the greedy algorithm is between 3 and 22%. Nevertheless, these results demonstrate satisfactory performance.

Network topology. We considered random graphs with 100 nodes and different number of edges. The goal is to test the accuracy of the algorithms for topologies with very different shapes, from a tree to a complete graph. We use here a connected variant of random Erdős-Rényi graphs. A graph is built as follows. We start from a random tree. An additional edge is present between two vertices u and v with probability p. For each experiment, we consider 400 random demands. We see, in Figure 5, that when the number of edges increases, the cost increases too. This is due to the fact that, when the number of edges increases, the average length of the shortest paths decreases. As discussed above, this reduces the opportunities of sharing. For small values of p, both algorithms have a similar accuracy. However, when $p \geq 0.25$, LP rounding provides optimal results in these settings.

Processing time. To study the limits in terms of computing time of an LP-based approach, we tested the LP-rounding and greedy algorithms using a larger topology: `Cogent` [19] with 197 nodes and 245 links. The algorithms have been implemented in C++, and the experiments were conducted on an Intel Xeon E5520 with 24GB of RAM. In Figure 6, we show the impact of the number of demands on the execution time. We compare the time necessary to find the optimal solutions with an ILP with the time needed by our algorithms to return a solution. We set a maximum time limit of one hour for each experiment. For just 500 demands, the time to find an exact solution exceeds 1 hour. This implies that, for large instances, an optimal solution cannot be found using the ILP in a reasonable amount of time. Both algorithms can compute solutions for larger instances. However, the greedy algorithm is much faster. Indeed, it takes 78 seconds to find a placement for 1200 demands, while the LP rounding algorithm requires more than 40

minutes.

7 Conclusion and Future Work

NFV is a novel approach for the deployment of network services that opens the way to a more efficient and flexible network management. Hence, placing network functions in a cost effective manner is an essential step toward the full adoption of the NFV paradigm.
In this paper, we investigated the problem of placing VNFs to satisfy the ordering constraints of the flows with the goal of minimizing the total setup cost. Since the formulated problem is NP-Hard, we proposed two algorithms that achieve a logarithmic approximation factor. To the best of our knowledge, no approximation algorithms have been proposed for the SFC Placement Problem in the literature so far. For the special case of tree network topologies with only upstream and downstream flows, we devised an optimal algorithm. Numerical results are given and validate the cost effectiveness of our algorithms.
This work aims at proposing a first theoretical framework for studying the placement problem with ordering constraints. However, a remaining unaddressed issue is considering flow rates and the accounting of practical constraints such as *soft capacities* on network functions or *hard capacities* on network nodes. An interesting future research direction may concern an investigation of the possibility of efficiently approximating these problems.

References

[1] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[2] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[3] M. Savi, M. Tornatore, and G. Verticale, "Impact of processing costs on service chain placement in network functions virtualization," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE, 2015, pp. 191–197.

[4] P. Quinn and T. Nadeau, "Problem statement for service function chaining," 2015.

[5] M. Obadia, J.-L. Rougier, L. Iannone, V. Conan, and M. Brouet, "Revisiting nfv orchestration with routing games," in *Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE Conference on*. IEEE, 2016, pp. 107–113.

[6] C. Chaudet, E. Fleury, I. G. Lassous, H. Rivano, and M.-E. Voge, "Optimal positioning of active and passive monitoring devices," in *Proceedings of the 2005 ACM conference on Emerging network experiment and technology*. ACM, 2005, pp. 71–82.

[7] I. Dinur and D. Steurer, "Analytical approach to parallel repetition," in *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '14. New York, NY, USA: ACM, 2014, pp. 624–633. [Online]. Available: http://doi.acm.org/10.1145/2591796.2591884

[8] M. C. Luizelli, L. R. Bays, L. Buriol, M. P. Barcellos, and L. P. Gaspary, "Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions," in *IFIP/IEEE International Symposium on Integrated Network Management*, 2015.

[9] T.-W. Kuo, B.-H. Liou, K. C.-J. Lin, and M.-J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," in *Computer Communications (INFOCOM), 2016 IEEE Conference on*. IEEE, 2016, pp. 1–9.

[10] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and placing chains of virtual network functions," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. IEEE, 2014, pp. 7–13.

[11] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on*. IEEE, 2015, pp. 1–6.

[12] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*. IEEE, 2015.

[13] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE, 2015, pp. 1346–1354.

[14] Y. Sang, B. Ji, G. R. Gupta, X. Du, and L. Ye, "Provably efficient algorithms for joint placement and allocation of virtual network functions," in *Computer Communications (INFOCOM), 2017 IEEE Conference on*. IEEE, 2017.

[15] G. Ausiello, A. D'Atri, and M. Protasi, "Structure preserving reductions among convex optimization problems," *Journal of Computer and System Sciences*, vol. 21, no. 1, pp. 136–153, 1980.

[16] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[17] N. Alon, D. Moshkovitz, and S. Safra, "Algorithmic construction of sets for k-restrictions," *ACM Trans. Algorithms*, vol. 2, no. 2, 2006.

[18] K. Menger, "Zur allgemeinen kurventheorie," *Fundamenta Mathematicae*, vol. 10, no. 1, pp. 96–115, 1927.

[19] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[20] V. V. Vazirani, *Approximation algorithms*. Springer Science & Business Media, 2013.

[21] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

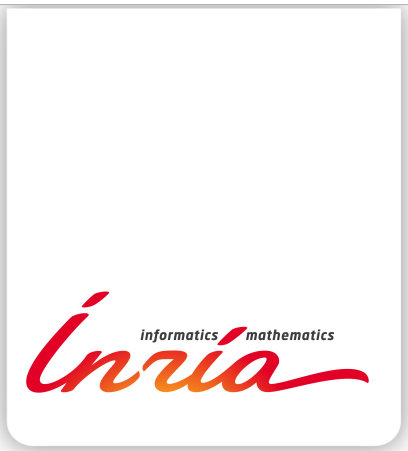
[22] K. Sohrabi, J. Gao, V. Ailawadhi, and G. J. Pottie, "Protocols for self-organization of a wireless sensor network," *IEEE personal communications*, vol. 7, no. 5, pp. 16–27, 2000.

[23] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li, "Design and deployment of a hybrid cdn-p2p system for live video streaming: experiences with livesky," in *Proceedings of the 17th ACM international conference on Multimedia*. ACM, 2009, pp. 25–34.

[24] I. Dinur and S. Safra, "On the hardness of approximating minimum vertex cover," *Annals of mathematics*, pp. 439–485, 2005.

[25] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski, "Sndlib 1.0—survivable network design library," *Networks*, vol. 55, no. 3, pp. 276–286, 2010.

[26] B. Bollobás, "Random graphs," in *Modern Graph Theory*. Springer, 1998, pp. 215–252.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399